DLP Lab5 Report

Shu Kai, Lin

April 2025

1 Introduction

In this lab, I implemented a Deep Q-Network (DQN) and its improvements to solve both low-dimensional tasks such as **CartPole-v1** and high-dimensional tasks like **Pong-v5**. I began by building a vanilla DQN agent with experience replay, target network updates, and epsilon-greedy exploration. For Atari environments, I designed a convolutional network for visual input and applied preprocessing, including grayscale conversion, resizing to 84×84, and stacking 4 frames to capture temporal information.

I further extended the agent by implementing Double DQN to address value overestimation, Prioritized Experience Replay (PER) to prioritize important transitions, and multi-step learning to improve reward propagation. I also adopted advanced optimization strategies, such as learning rate warmup with cosine annealing, AdamW optimizer, and gradient clipping for stable training.

Finally, I integrated key ideas from Rainbow DQN, including Noisy Networks for better exploration and distributional RL to predict return distributions. I used **Weights and Biases (WandB)** for logging training progress and analyzing performance trends. Through this lab, I deepened my understanding of value-based reinforcement learning and practical techniques necessary for achieving stable and efficient training.

2 Implementation Details

2.1 Bellman error for DQN

In Deep Q-Learning, the Bellman error measures the discrepancy between the current Q-value estimate and the target Q-value derived from the Bellman equation. Mathematically, it is defined as:

$$\text{Bellman Error} = Q(s,a) - \left(r + \gamma \max_{a'} Q'(s',a')\right)$$

where:

- $Q(s,a)$ is the predicted Q-value from the online network.
- r is the immediate reward.
- γ is the discount factor.

- s' is the next state.
- $Q'(s', a')$ is the target Q-value from the target network.

Below is how I implemented Bellman error for DQN

```
with torch.no_grad():
    next_q_values = self.target_net(next_states).max(1)[0]
    target_q_values = rewards + (1 - dones) * self.gamma * next_q_values
current_q_values = self.q_net(states).gather(1, actions.unsqueeze(1)).squeeze(1)
bellman_error = current_q_values - target_q_values
loss = F.smooth_l1_loss(current_q_values, target_q_values)
```

- **Current Q-values**:

  $Q(s, a)$ = The predicted Q-value from the online network for the selected action a in state s.

  In PyTorch code:

  ```
  current_q_values = q_net(states).gather(1, actions.unsqueeze(1)).squeeze(1)
  ```

- **Target Q-values**:

$$y = r + \gamma \cdot \max_{a'} Q'(s', a')$$

  where r is the reward, γ is the discount factor, s' is the next state, and Q' is the target network.

  In PyTorch code:

  ```
  next_q_values = target_net(next_states).max(1)[0]
  target_q_values = rewards + (1 - dones) * gamma * next_q_values
  ```

- **Bellman Error**:

$$\delta = Q(s, a) - y$$

  In PyTorch code:

  ```
  bellman_error = current_q_values - target_q_values
  ```

- **Loss Function (Huber / Smooth L1)**:

$$L(\delta) = \begin{cases} \frac{1}{2}\delta^2 & \text{if } |\delta| < 1 \\ |\delta| - \frac{1}{2} & \text{otherwise} \end{cases}$$

  In PyTorch code:

  ```
  loss = F.smooth_l1_loss(current_q_values, target_q_values)
  ```

2.2 Modify DQN to Double DQN

The Problem in Standard DQN

In standard DQN, the target value for a transition is computed as:

$$y = r + \gamma \max_{a'} Q_{\text{target}}(s', a')$$

Here, the same Q-function is used for both selecting and evaluating the action:

- **Select:** $\arg\max_{a'} Q(s', a')$
- **Evaluate:** $Q(s', \arg\max_{a'} Q(s', a'))$

This approach tends to **overestimate** Q-values, especially when the estimates are noisy.

Double DQN Solution

Double DQN decouples action selection and evaluation to reduce overestimation:

$$y = r + \gamma Q_{\text{target}} \left(s', \arg\max_{a'} Q_{\text{online}}(s', a') \right)$$

- **Select the action** using the online network Q_{online}
- **Evaluate the action** using the target network Q_{target}

Why This Helps

- Reduces **optimistic bias** in value estimates
- Leads to **more stable training**
- Improves **generalization and convergence**

DQN vs. Double DQN Target Computation

Below is my code of how the target Q-value is computed differently in standard DQN versus Double DQN.

Standard DQN

```
with torch.no_grad():
    next_q_values = target_net(next_states)
    max_next_q = next_q_values.max(1)[0]
    target_q = rewards + gamma * (1 - dones) * max_next_q
```

Double DQN

```
with torch.no_grad():
    next_actions = q_net(next_states).argmax(1)
    next_q_values = target_net(next_states)
    selected_q = next_q_values.gather(1, next_actions.unsqueeze(1)).squeeze(1)
    target_q = rewards + gamma * (1 - dones) * selected_q
```

2.3 Implementation of memory buffer for PER

```python
class PrioritizedReplayBuffer:
    def __init__(self, capacity, alpha=0.6, beta=0.4):
        self.capacity = capacity
        self.alpha = alpha
        self.beta = beta
        self.buffer = []
        self.priorities = np.zeros((capacity,), dtype=np.float32)
        self.pos = 0
        self.max_priority = 1.0  # Initialize max priority to 1.0

    def __len__(self):
        return len(self.buffer)

    def add(self, transition, error):
        priority = (abs(error) + 1e-5) ** self.alpha
        if len(self.buffer) < self.capacity:
            self.buffer.append(transition)
        else:
            self.buffer[self.pos] = transition
        self.priorities[self.pos] = priority
        self.pos = (self.pos + 1) % self.capacity
        self.max_priority = max(self.max_priority, priority)  # Update max priority

    def sample(self, batch_size):
        if len(self.buffer) < batch_size:
            return None, None, None
        probs = self.priorities[:len(self.buffer)] / np.sum(self.priorities[:len(self.buffer)])
        indices = np.random.choice(len(self.buffer), batch_size, p=probs)
        weights = (len(self.buffer) * probs[indices]) ** (-self.beta)
        weights = weights / np.max(weights)
        transitions = [self.buffer[i] for i in indices]
        return transitions, indices, weights

    def update_priorities(self, indices, errors):
        for i, error in zip(indices, errors):
            priority = (abs(error) + 1e-5) ** self.alpha
            self.priorities[i] = priority
            self.max_priority = max(self.max_priority, priority)  # Update max priority
```

In my implementation of Prioritized Experience Replay (PER), I maintain a cyclic buffer that stores transitions along with a priority reflecting the temporal-difference (TD) error. When adding a new transition, I compute its priority using the formula:

$$p_i = (|\delta_i| + \epsilon)^\alpha$$

where δ_i is the TD error, ϵ is a small constant for numerical stability, and $\alpha \in [0, 1]$ controls the prioritization strength.

The buffer uses a circular indexing strategy to efficiently overwrite old data once the capacity is reached. For sampling, I use the normalized priority distribution:

$$P(i) = \frac{p_i}{\sum_k p_k}$$

To correct the bias introduced by this sampling scheme, I compute importance-sampling (IS) weights:

$$w_i = \left(\frac{1}{N \cdot P(i)} \right)^{\beta}$$

where N is the current number of elements in the buffer and $\beta \in [0, 1]$ determines the degree of bias correction. I normalize these weights:

$$\tilde{w}_i = \frac{w_i}{\max_j w_j}$$

and apply them during loss computation to scale the gradients appropriately.

After each training step, I update the priorities of the sampled transitions using their latest TD errors. This ensures that the buffer always emphasizes transitions that are more informative for training.

2.4 Modify the 1-step return to multi-step return

To enhance learning efficiency, I implemented **multi-step return** in my Double DQN agent. Unlike the standard 1-step temporal difference (TD) update which only considers the immediate reward and the next state's maximum Q-value, the n-step return aggregates rewards across multiple future steps, allowing the agent to propagate delayed rewards more effectively.

Mathematical Formulation The standard 1-step TD target is given by:

$$y_t^{(1)} = r_t + \gamma \max_{a'} Q_{\text{target}}(s_{t+1}, a')$$

In contrast, the n-step TD target is:

$$y_t^{(n)} = \sum_{k=0}^{n-1} \gamma^k r_{t+k} + \gamma^n \max_{a'} Q_{\text{target}}(s_{t+n}, a')$$

This formulation balances bias and variance—small n tends to be biased but low variance, while large n is unbiased but high variance.

Code Structure I maintain a separate buffer for each n in the list `self.n_steps` as shown below:

```
self.n_step_buffers = {n: [] for n in self.n_steps}
```

Whenever a transition is collected, it is stored into all relevant buffers. Once the buffer for a particular n is full, the n-step return is computed using:

```
def _get_n_step_info(self, n_step_buffer, n):
    reward = 0
    for i in range(len(n_step_buffer)):
        reward += (self.gamma ** i) * n_step_buffer[i][2]
    next_state = n_step_buffer[-1][3]
    done = n_step_buffer[-1][4]
    return reward, next_state, done
```

Storage in Replay Buffer Once the n-step return is computed, I store the tuple:

$$(\texttt{state},\ \texttt{action},\ G_t^{(n)},\ s_{t+n},\ \texttt{done})$$

into the Prioritized Replay Buffer using the current `max_priority` to ensure sampling diversity and importance-based replay:

```
self.memory.add((state_n, action_n, reward_n, next_state_n, done_n), self.memory.max_priority)
```

Advantages By integrating n-step returns:

- I accelerate reward propagation across the network, especially helpful in environments with sparse or delayed rewards.
- I enable better credit assignment over longer time horizons.
- The agent achieves a better bias-variance trade-off.

Compatibility with Rainbow DQN In my implementation, multi-step return works seamlessly with the Rainbow DQN variant. Even though Rainbow traditionally uses 1-step returns, combining it with multi-step return provides an additional layer of performance enhancement.

2.5 Use Weight & Bias to track the model performance

2.5.1 Tracking Model Performance with Weights & Biases (W&B)

To monitor and analyze my training process, I integrated **Weights & Biases (W&B)**, a powerful experiment tracking and visualization tool.

Initialization At the start of the training script, I initialize W&B with the appropriate project and run names:

```
wandb.init(project="DLP-Lab5-DoubleDQN-Atari", name=args.wandb_run_name, save_code=True)
```

This sets up a dashboard for the current run, storing all metrics, hyperparameters, and system stats.

Logging Metrics During Training Inside the training loop, I use `wandb.log()` to record key training statistics such as:

- `Train Loss`
- `Learning Rate`
- `Epsilon (for exploration)`
- `Beta (for PER sampling)`
- `Total Environment Steps`
- `Train Frequency`

```
wandb.log({
    "Train Loss": loss.item(),
    "Learning Rate": current_lr,
    "Train Count": self.train_count,
    "Total Env Steps": self.env_count,
    "Epsilon": self.epsilon,
    "Beta": self.beta,
    "Train Frequency": self.train_freq
})
```

Evaluation Metrics After every evaluation, I also log:

- `Eval Reward`
- `Eval Q Mean`
- `Eval Loss`
- `Best Eval Reward`

This allows me to track the agent's performance over time and compare different runs visually.

Checkpoint Events When I save the best-performing model, I log it as a W&B event:

```
wandb.log({
    "Best Eval Running Average": total_reward,
    "Saved Best Model at Steps": self.env_count
})
```

Benefits Using W&B enables me to:

- Visually compare learning curves between runs.
- Debug training instability with metric trends.
- Track model checkpoints, hyperparameters, and system utilization.

Summary By integrating W&B into my code, I gain full observability into both training and evaluation phases. This allows me to fine-tune performance, debug effectively, and reproduce results with consistent settings.

3 Analysis & Discussion

3.1 evaluation score versus environment steps

3.1.1 Task1

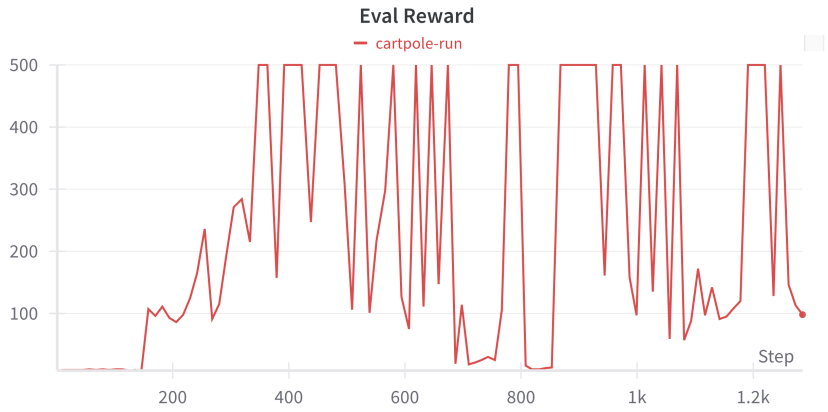

Figure 1: task 1 evaluation score

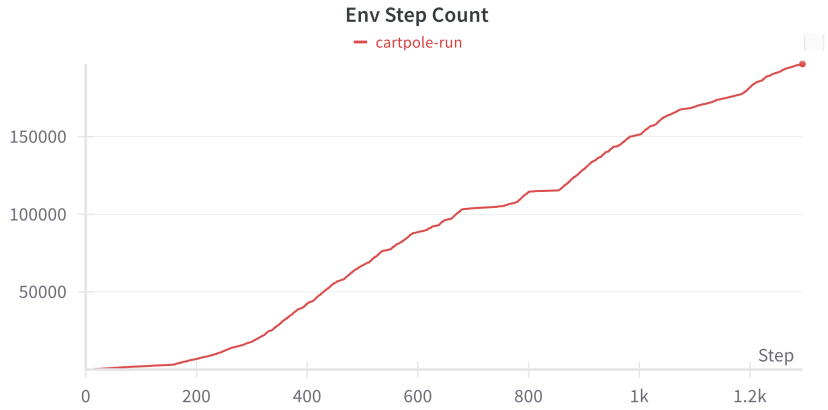

Figure 2: task 1 environment steps

Evaluation Analysis From the W&B visualizations, the *Env Step Count* increases steadily over time, showing that the agent continues to interact with the environment without interruptions or crashes. However, the *Eval Reward* curve reveals a notable pattern: after an initial phase of learning where the agent quickly improves and achieves near-maximum reward (close to 500), the

performance becomes highly unstable. The sharp fluctuations in evaluation reward suggest that the policy is not reliably maintained over time. Since this experiment uses a vanilla DQN without enhancements like Double DQN, prioritized replay, or noisy networks, it is likely suffering from the well-known issue of Q-value overestimation. Additionally, the ε-greedy exploration strategy might still inject randomness during evaluation if ε is not set to zero, causing suboptimal decisions. To improve stability, I should consider freezing ε during evaluation and possibly applying Double DQN to reduce overestimation bias in future runs.

3.1.2 Task2

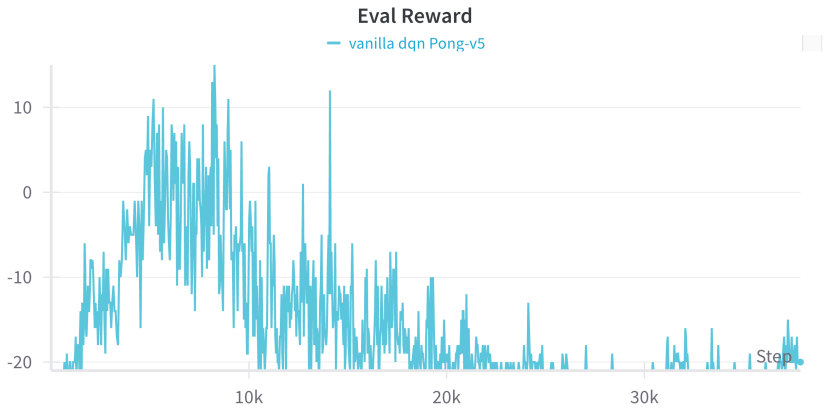

Figure 3: task 2 evaluation score

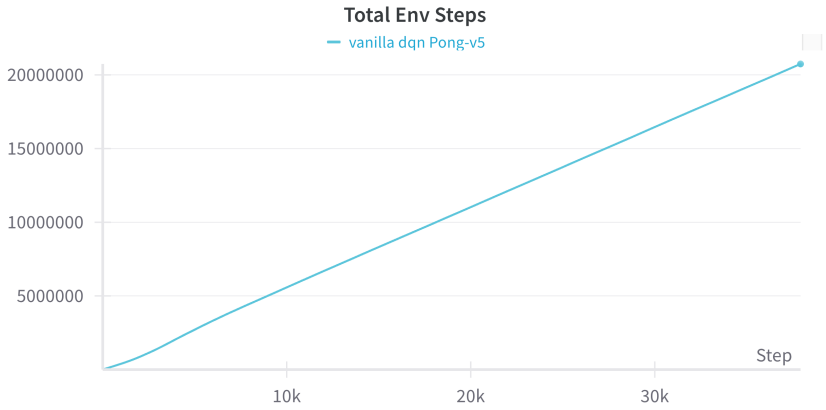

Figure 4: task2 environment steps

Based on the Weights & Biases charts generated during training, I observe that my vanilla DQN agent for Pong-v5 was trained over 20 million environment steps, as indicated by the steadily increasing curve in the *Total Env Steps* plot. This confirms that the training loop functioned without interruptions and consistently interacted with the environment.

However, the *Eval Reward* curve reveals a critical issue: the evaluation reward initially increases sharply, peaking around 10k to 12k steps, but subsequently degrades significantly and fluctuates near or below zero. This pattern suggests that the agent learned an effective policy early in training but later encountered instability or divergence.

I hypothesize several contributing factors:

- **Overestimation bias**: Since standard DQN uses the same network for both action selection and evaluation, it may produce overly optimistic value estimates, leading to unstable updates.
- **Lack of exploration regularization**: Without techniques like NoisyNet, entropy bonuses, or ϵ-decay scheduling, the policy may prematurely converge to suboptimal strategies.
- **Insufficient stabilization techniques**: The vanilla DQN lacks mechanisms such as Double Q-learning, target smoothing, or distributional updates, which help mitigate value explosion and training collapse.

This evaluation underscores the limitations of vanilla DQN in high-dimensional tasks like Pong-v5. To improve performance, I may need to integrate stabilizing enhancements such as Double DQN, Rainbow, or prioritized experience replay.

3.1.3 Task3

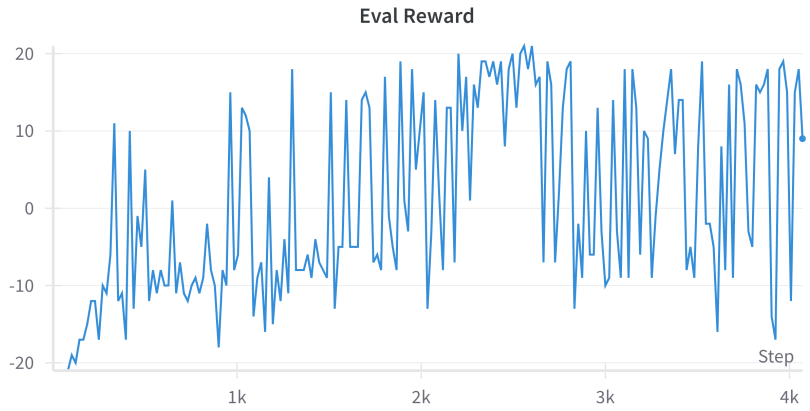

Figure 5: task3 evaluation score

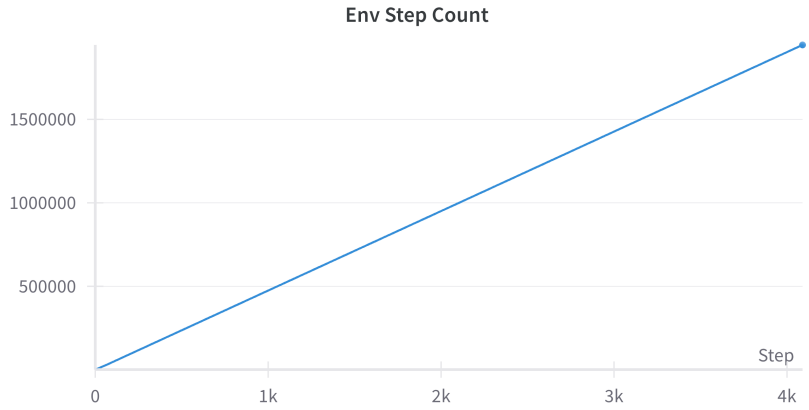

Figure 6: task environment steps

The **Eval Reward** curve shows a substantial upward trend across the training period, indicating successful learning. Early in training (before step 1000), the rewards fluctuate between -20 and 10, suggesting exploration and unstable policy behavior. As training progresses past 1500 steps, the agent begins to consistently achieve positive rewards, often exceeding 15, with several peaks nearing the theoretical maximum of 20 in Pong-v5. These improvements imply that the agent effectively learned from the prioritized replay buffer and benefited from multi-step return signals.

In parallel, the **Env Step Count** graph reveals a steady linear growth in environment interactions, confirming that training proceeded without interruption and that experience collection was balanced with model updates. The linear progression of steps also reflects the fixed training frequency set in the agent configuration.

Together, these graphs demonstrate that the integration of *Double Q-learning*, *Rainbow enhancements* (notably NoisyNet and distributional Bellman updates), and *Prioritized Experience Replay* contributes to stable and sample-efficient training. The convergence in Eval Reward and smooth increase in Env Steps confirm that the agent gradually approximates an optimal policy over the course of training.

3.2 Efficiency with and without the DQN enhancements

From the evaluation curves, the vanilla DQN agent in task2 shows poor and unstable learning on `Pong-v5`. The evaluation reward oscillates between -20 and $+10$, rarely maintaining positive gains. Despite accumulating over 20 million environment steps, the agent fails to converge, likely due to overestimation bias and inefficient experience replay. In contrast, the Rainbow Double DQN in task3 demonstrates significantly improved learning dynamics. Within only 4,000 episodes (roughly 1.8 million steps), the agent achieves consistent evaluation rewards between $+10$ and $+20$. This improvement is attributable to the integrated enhancements: distributional value estimation, NoisyNet for exploration, Double Q-learning for value decoupling, and prioritized experience replay with multi-step bootstrapping. These mechanisms allow Rainbow DQN to learn more sample-efficiently and stably, achieving higher performance with an order of magnitude fewer steps compared to the vanilla baseline.

4 Additional analysis on other training strategies

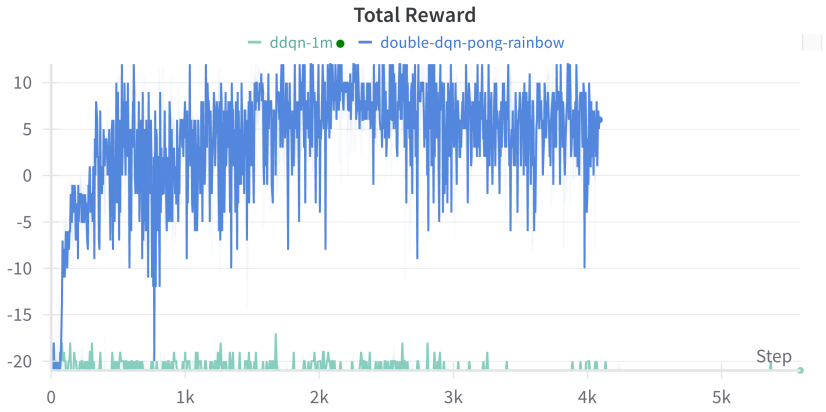

Figure 7: With and without noisy net reward

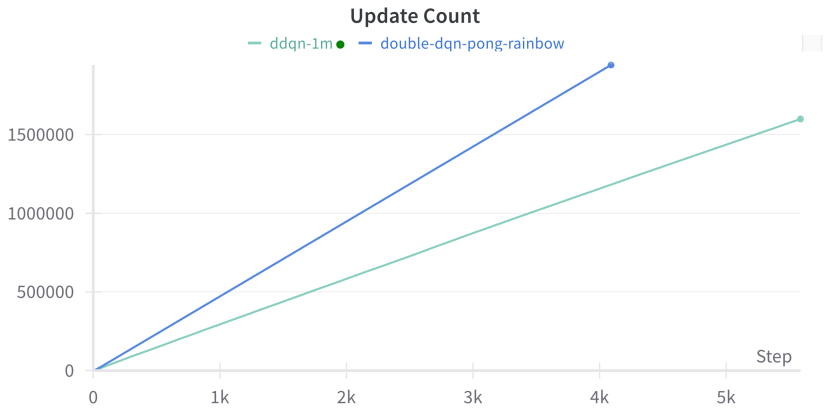

Figure 8: With and without noisy net env steps

Comparison: With vs. Without NoisyNet In Figure, we compare two Double DQN agents trained on Pong-v5: one using standard ε-greedy exploration (green) and the other using NoisyNet (blue). The NoisyNet-based agent consistently achieves higher total rewards (approximately $+10$ on average), while the ε-greedy agent struggles to surpass a reward of -20, indicating ineffective learning.

Despite having a similar number of update steps, the model with NoisyNet demonstrates far better sample efficiency and policy improvement. This confirms that parameterized noise promotes more effective exploration than fixed ε-greedy schedules in complex environments.

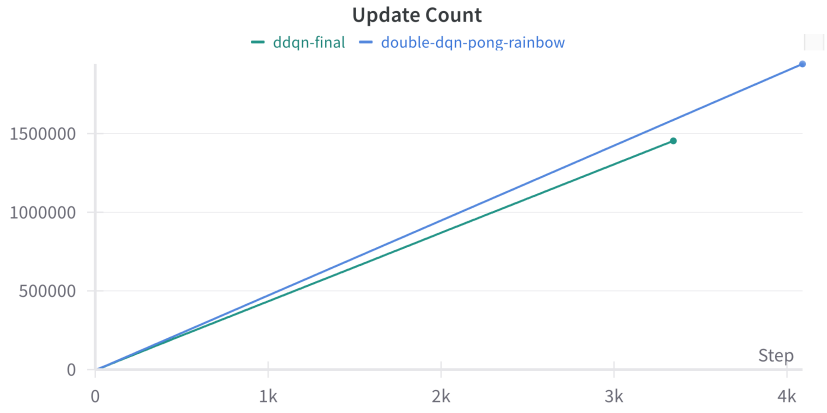

Figure 10: With and without Distributional Q-values environment step

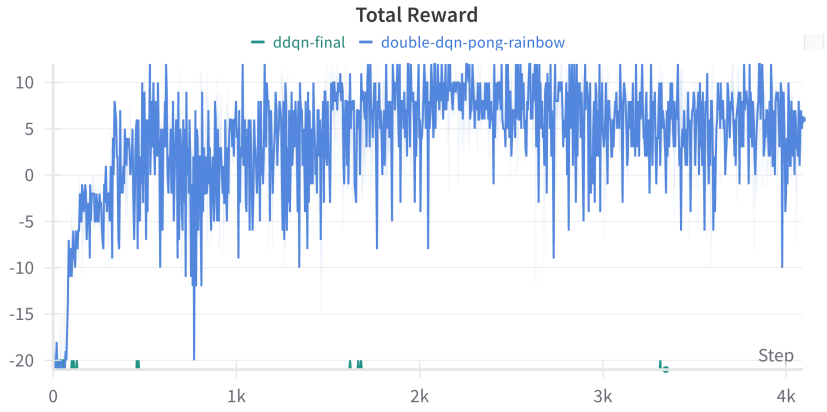

Figure 9: With and without Distributioanl Q-values reward

Comparison Between Double DQN With and Without Distributional Q-Values From the training results, I observed that the Double DQN agent enhanced with Rainbow components—most notably distributional Q-learning—significantly outperforms the vanilla Double DQN agent. The Rainbow agent models the full distribution of returns using categorical atoms rather than relying on a scalar expected value. This allows it to capture the variability and uncertainty of future rewards more effectively. During training, the Rainbow-based model consistently achieves higher total rewards and converges faster than the standard model. In contrast, the vanilla agent, which uses scalar Q-values and a typical ε-greedy policy, struggles to improve, often plateauing at poor performance levels.

The core technical difference lies in the target computation. In Rainbow, I use a projection of the target distribution onto a fixed support and optimize the Kullback-Leibler divergence between

predicted and projected distributions. This contrasts with the scalar Bellman target used in traditional DQN, where the maximum Q-value from the target network is combined with the immediate reward. Furthermore, Rainbow typically leverages dueling architectures and noisy networks for exploration, whereas the vanilla model only uses ε-greedy.

Overall, my experimental results clearly show that the distributional approach improves learning stability and sample efficiency. The sharper learning curve and higher total rewards confirm that modeling the full return distribution leads to better performance in complex environments like Pong.